FlightLinux Project

Bulk Memory Device Driver
Concept

Technical Report

March 30, 2001

Draft of March 27

Patrick H. Stakem
QSS Group, Inc.

Introduction

This report is one of a series of technical reports by the FlightLinux Project, and covers the definition of the software device driver for the bulk memory component. The goal is to abstract the bulk memory via software into a device that the operating system sees and treats as a disk file system. This discussion will focus on the specific requirements of the Surrey Space Technology Lab (SSTL) UoSat-12 mission, which will be the pathfinder flight for FlightLinux. Because the bulk memory is expected to be susceptible to many environmentally induced errors, an approach will be defined that will allow adequate error management. At the same time, the device driver may be abstracted from the specifics of the UoSat-12 implementation, and used for other missions.

Background

This work was conducted under task NAS5-99124-297, with funding by the NASA Advanced Information Systems Technology (AIST) Program, NRA-99-OES-08. The work is conducted by personnel of QSS Group, Inc. in partnership with NASA/GSFC codes 586 (Science Data Systems) and code 582 (Flight Software).

The FlightLinux project has the stated goal of providing an on-orbit flight demonstration of the software within the contract period. Numerous other Linux efforts exist within the GSFC software community The FlightLinux Project has memoranda of understanding with the OMNI (Operating Missions as Nodes on the Internet) Project at GSFC, and with Surrey Space Technology Labs. SSTL will allow the use of the in-orbit UoSat-12 spacecraft for FlightLinux testing.

Spacecraft onboard computers do not usually employ rotating magnetic memory for secondary storage. Initially, magnetic tape was used, but now the state of the art is to use large arrays of bulk DRAM, with various error detection and correction hardware and/or software applied.

Device driver

A device driver is a low level software routine that interfaces hardware to the operating system. It abstracts the details of the hardware, such that the operating system can deal with a standardized interface for all devices. In Unix-type operating systems such as Linux, the file system and the I/O devices are treated similarly.

Device drivers are prime candidates for implementation in assembly language, because of the need for bit manipulation, and for speed. They can also be implemented in higher-order languages such as c, however. Typical device drivers would include those for serial ports, for parallel ports, for the mass storage interface (SCSI, for example), for the lan interface, etc.

Device drivers are both operating system-specific, and specific to the device being interfaced. They are custom code, used to adapt and mediate environments.

Bulk memory

As secondary storage for telemetry, the previous choice was magnetic tape. The current state-of-the-art is bulk memory; essentially large blocks of dynamic random access memory (DRAM). This memory, usually still treated as a sequential access device, is mostly used to hold telemetry during periods when ground contact is precluded. Bulk memory is susceptible to errors on read and write, especially in the space environment, and needs multi-layer protection such as triple-modular redundancy (TMR), horizontal and vertical cyclic redundancy codes (CRC), error correcting codes (ECC), and scrubbing. Scrubbing can be done by hardware or software in the background. The other techniques are usually implemented in hardware. With a memory management unit (MMU), even using 1:1 mapping of virtual to physical addresses, the MMU can be used to re-map around failed sections of memory.

Although we usually think of bulk memory as a secondary storage device with sequential access, it may be implemented as random access memory within the computer's address space. This is the case with UoSat-12.

The Flash File System Model

The flash file system (FFS) has been developed to treat collections of flash memory as a disk drive, with an imposed file system. Although we are dealing with DRAM and not flash, we can still gain valuable insight from the FFS implementation. In addition, the implementation of Linux support for the pcmcia devices (ref. 6) provides a useful model.

The bulk memory on UoSat-12

The onboard computers on the UoSat-12 spacecraft has 128 megabytes of DRAM bulk memory. It is divided into 4 banks of 32 megabytes each, mapped through a window at the upper end of the processor's address space. This is the specific device driver that we will develop, and use as a model for future development of similar modules. The current software of the UoSat-12 onboard computer treats this bulk memory as paged random access memory, and applied a scrubbing algorithm to counter environmentally induced errors.

The ramdisk under Linux

The ramdisk is a disk-like block device implemented in ram. This is the correct model for using the bulk memory of the onboard computer as a file system. Multiple ramdisks may be allocated in Linux. The standard Linux utility mke2fs, which creates a Linux second extended file system, works with ramdisk, and supports RAID level 0.

The RAID model

The RAID (redundant arrays of independent disks) model was developed to use large numbers of commodity disk drives combined into one large, fault-tolerant storage unit. The approach can be applied to bulk memory as well. RAID can be implemented in software or hardware. For the purposes of this document, we will consider RAID software implementations. Software RAID is a standard Linux feature, available as a patch to the 2.12 kernels, and slated to become an included feature in kernel 2.4.

The following definition is abstracted from reference 3:

The term "Redundant Array of Inexpensive Disks" or RAID first appeared in a University of California at Berkeley white paper published in 1987. The paper addressed the topic of how to organize several small disks in various fashions to appear as one large drive, to increase data availability and performance. The white paper described the status of the various RAID work that had been done to date, and labeled the existing RAID levels as 1 through 5.

RAID technology involves the use of electronic controller hardware or software, or a combination of both, to connect disk drives to computer systems in such a way that when a physical disk drive fails or "crashes," the data on that drive is not lost but can be retrieved by "reconstruction" using parity data. Seven such basic connection schemes or architectures have been devised. All RAID product levels on the market are proprietary implementations; however, most vendors adhere to the industry standard SCSI protocols of the ANSI X3.131 and ISO/IEC standards. The ANSI/ISO/IEC standards provide RAID customers open system integration, portability, and investment protection.

RAID 0 offers disk striping without parity. The multiple disks provide quick reads and writes for large files without the data redundancy protection provided by parity. Level 0 is not considered true RAID.

RAID Level 1

All data is copied onto two separate disks for full data redundancy. This technique has been used in various systems since the 1960s and is often referred to as disk mirroring, or dual copy, or disk shadowing in a subsystem. The subsystem controller(s) write data simultaneously to both disk sets, which means that writes are nearly as fast as to a single disk, and reads can be done from the disk set that offers the fastest read. Level 1 is a fast, secure and reliable form of RAID. The disadvantage is in overhead because twice as many physical drives are required.

RAID 2 is a patented architecture of Thinking Machines Inc. that uses multiple dedicated parity disks in a Hamming Code scheme which in turn requires that the array be built with a relatively large number of disk drives. The parity disks avoid the cost of full duplication of the disk drives of Level 1. If a disk fails, parity reconstructs data without system loss. In Level 2, all the disks in the array are synchronized, which means that all disks must be accessed in parallel. This is advantageous in applications that require very large contiguous file transfers at high bandwidth but ineffective for applications requiring many small reads and writes. For these reasons, Level 2 is not commercially viable.

Like Level 2, RAID Level 3 has synchronized disks across which data is bit or byte striped, making Level 3 effective for large data transfers-as in imaging or some scientific or technical applications-but not for small transactions. In Level 3, all parity data is stored on a single parity drive, thus reducing the cost but without a performance improvement in a highly interactive computing environment like UNIX. There are many proprietary implementations of Level 3 available.

Like Level 3, RAID Level 4 uses a single parity drive with data striped in sectors or blocks, but disks in the array are not synchronized, meaning multiple reads to disks can be done independently. While this improves performance for many small reads, writes still are a drawback. Since the parity drive must be updated for each write operation, all drives must wait until the updated user data and its parity are complete, causing a bottleneck at the parity drive. Although some proprietary Level 4 products were introduced, it appears that RAID Level 4 is not commercially viable.

Like Level 4, RAID Level 5 allows independent disk access, but parity data is spread across all the disks in the array. Distributed parity defers the parity disk bottleneck problem. Write performance is improved, but three write operations (read-verify-write) still are required to read and update the old parity with two drives locked in parallel. This is called the Level 5 write penalty. Since reads normally are far more frequent than writes, performance is improved over Level 4 and 3, but is still not equal to the speed of disk mirroring solutions. However, the capacity penalty for parity data in Level 5 is typically 20% compared to 100% for Level 1, and therefore there are many proprietary implementations of Level 5 commercially available.

(end abstract)

The technique of RAID is not limited to the implementation of rotating magnetic media; we could as well say that RAID also stands for Redundant Array of Inexpensive DRAM. We will explore several techniques for implementation, parallel to RAID(isk) level 1 and 5. One key point is that the RAID technique will make the accesses slower than with standard DRAM, but still much faster than achievable on a mechanical rotating system, or a sequential access system such as tape.

Bulk Memory Device Driver Level 0

This initial version of the driver will use memory mirroring, with memory scrubbing techniques applied. In the simplest case, we will treat three of the four available 32-

megabyte memory pages as a mirrored system. The memory scrubbing technique will be derived from the current scheme used by SSTL, as will the paging scheme.

Bulk Memory Device Driver Level 1

This version of the driver will use all four of the available 32-megabyte memory pages with distributed parity. The performance with respect to write speed is expected to be less than with the Level 0, but the memory resilience with respect to error should be much better.

Testing

It is unclear without further testing whether the RAID technique will be sufficient to counter the environmentally-induced errors expected in the bulk memory on-orbit. It is generally accepted that RAID is not intended to counter data corruption on the media, but rather to allow data recovery in case of media failure. The testing approach outlined in Appendix A will be used with the bulk memory device driver on the breadboard facility. More extensive testing on-orbit with the UoSat-12 spacecraft will be required to validate the approach.

Appendix A - Test Methodology for M-disk RAID device

by J. Todd Miller, GSFC Code 582

This approach uses a "side-band" which has direct access to the RAM which comprises the disks. Assuming that the RAM disks are just normal block devices, it should be possible to corrupt them by accessing them directly with a root privileged task: i.e.

-1. Set up RAM disk devices.
0. Setup RAID and write a test pattern to the RAID device to be verified later.

1. Open a ram disk as a file (assuming that the kernel lets you; you should be root).
2. Seek a random spot on the ram disk file.
3. Read some data.
4. Mask in an error pattern. Start with single bit errors. Then try double bit errors in a single byte.
5. Seek the same random spot.
6. Write the data back.
7. Close the ram disk file.

8. Access the RAID device to "discover" the error.

This whole process can be done with loopback files rather than RAM disks, enabling you to set up the RAID elements as persistent things. This persistence can be exploited if the kernel decides to "get in the way" and say "device busy" when you try to access a RAID element. Then the scenario might be:

-1. Set up loopback devices.
0. Set up RAID array on loopback devices.
1. Deactivate RAID array.
2. Corrupt loopback device file.
3. Reactivate RAID array.
4. Verify test pattern by accessing RAID device.

References

1. Project website at http://FlightLinux.GSFC.NASA.GOV

2. Rubini, Alessandro: *Linux Device Drivers*, 1st Edition, February 1998, ISBN 1-56592-292-1

3. http://bart.intel-sol.com/products/scc/print.wp_raid7.html

4. Linux Software RAID HOWTO

5. Linux ramdisk.txt (/usr/src/linux/Documentation)

6. Linux PCMCIA Programmers Guide, http://cnuce-arch.cnuce.cnr.it/Linux/kernel/pcmcia/doc/PCMCIA-PROG.html

7. patch-2.3.19 linux/drivers/pcmcia/bulkmem.c http://www.kernelnotes.org/v23patch/patch2.3.19/linux_drivers_pcmcia_bulkmem.c.html